VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

APLIKACE PRO DETEKCI ZRANITELNOSTI TYPU CLICKJACKING

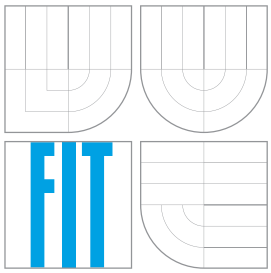
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR STODŮLKA

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

APLIKACE PRO DETEKCI ZRANITELNOSTI TYPU CLICKJACKING

TOOL FOR CLICKJACKING VULNERABILITY DETECTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PETR STODŮLKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. MAROŠ BARABAS
SUPERVISOR

BRNO 2013

Abstrakt

Tato bakalářská práce se zabývá zranitelností známou jako Clickjacking nebo také UI redressing. Podrobně je zde popsáno nebezpečí, které tato zranitelnost přináší, spolu s možnostmi zabezpečení, které jsou dnes známy. Cílem práce je vytvoření aplikace detekující přítomnost zabezpečení těchto zranitelností.

Abstract

This bachelor's thesis discusses a vulnerability known as Clickjacking, or UI redressing. The thesis describes in detail the dangers, to which this vulnerability may lead, and presents known countermeasures. The goal of this thesis is to create an application which detects the presence of these countermeasures.

Klíčová slova

Clickjacking, UI redressing, XSS, bezpečnost webových aplikací, penetrační testování, HTML, IFRAME, CSS, Javascript, X-Frame-Options, CSP

Keywords

Clickjacking, UI redressing, XSS, web security, penetration testing, HTML, IFRAME, CSS, Javascript, X-Frame-Options, CSP

Citace

Petr Stodůlka: Aplikace pro detekci zranitelnosti typu Clickjacking, bakalářská práce, Brno, FIT VUT v Brně, 2013

Aplikace pro detekci zranitelnosti typu Clickjacking

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Maroše Barabase

.....
Petr Stodůlka
15. května 2013

Poděkování

Rád bych poděkoval vedoucímu mé práce, panu Ing. Marošovi Barabasovi, za odborné rady, připomínky k mé práci a čas který mi věnoval. Rovněž bych chtěl také poděkovat panu Ing. Michalu Drozdovi, pod kterým jsem tuto práci začal původně vytvářet, za předchozí vedení a odborné rady, které mi poskytl.

Obsah

Kapitola 1

Úvod

V posledních letech se internet stává součástí každodenního života. S narůstajícím počtem uživatelů nabývá na stále větším významu otázka bezpečnosti služeb poskytovaných na webu a ochrana soukromí jejich uživatelů. Webové aplikace dnes musí čelit stále se rozrůstající řadě možných útoků, které využívají jejich zranitelností.

Jedním z novějších útoků využívá zranitelnosti typu *clickjacking*[1]. Clickjacking byl prezentován v roce 2008 bezpečnostními specialisty Jeremiah Grossmanem a Robertem Hansenem [11]. Ve stejném období prezentoval tento typ útoku také Michal Zalewski pod pojmem *UI redressing* [13]. Tento pojem se však neuchytil.

Nebezpečí clickjackingu spočívá v širokém spektru možností, které poskytuje. Lze jej využívat k úspěšnému provedení dalších útoků, proti kterým by byla webová aplikace jinak zabezpečená. Jeho prostřednictvím lze nechat uživatele provádět úkony pod vlastní identitou, které by jinak neprovedl. Může být využit k získání privátních informací uživatelů a v neposlední řadě lze jeho užitím připravit uživatele o finanční prostředky, pokud není proti tomuto útoku internetové bankovnictví zabezpečeno a uživatel je přihlášen. A to vše lze nechat uživatele provést, aniž by cokoliv tušil.

Dalším důvodem, proč je clickjacking natolik úspěšný, je nízké povědomí o této zranitelnosti mezi vývojáři webových aplikací a tedy nízký počet takto zabezpečených aplikací. Z mé osobní zkušenosti drtivá většina vývojářů webových aplikací o existenci této zranitelnosti nevěděla, ať už se jednalo o zkušené vývojáře s letitou praxí, kteří měli povědomí o nejběžnějších zranitelnostech a řešili je ve svých aplikacích, či začínající programátory.

1.1 Cíle práce

Hlavním cílem této bakalářské práce je vytvoření aplikace, která bude detekovat výskyt zranitelnosti typu clickjacking ve webové aplikaci. Aplikace by měla být vyvíjena jako nástroj určený k provádění *penetračních testů*. Nástroje schopné detekce této zranitelnosti jsou převážně vyvíjeny jako přídavné moduly webových prohlížečů a jsou určeny především k zastavení právě probíhajícího pokusu o provedení clickjackingu nebo jsou na prohlížečích závislé[2].

Další motivací k vytvoření této práce je pro mne možnost přispět alespoň minimálně k rozšíření povědomí o zmiňované zranitelnosti mezi vývojáři webových apikací i obyčejnými uživateli, kteří si v budoucnu tuto práci přečtou.

[1] Clickjacking bude podrobně vysvětlen v kapitole 4

[2] Viz 4.2.4 a 4.3.

1.2 Struktura práce

Kapitola 2 obsahuje stručný popis HTTP protokolu.

V kapitole 3 jsou blíže vysvětleny bezpečnostní politiky SOP a CSP a popis XSS útoků. V sekci 3.4 je zde vysvětlena potřeba penetračního testování spolu se dvěmi vybranými penetračními nástroji Paro a BurpSuite.

Clickjackingu je věnována kapitola 4, kde je v úvodu provedena podrobná jeho analýza. V sekci 4.2 jsou rozebrány způsoby zabezpečení webové aplikace spolu s možnostmi obrany ze strany uživatele. Na závěr jsou zde rozebrány možná způsoby detekce zranitelnosti a popis jiného nástroje, který se automatizovanou detekcí clickjackingu zabývá.

Vytyčené požadavky na vlastní aplikaci jsou rozepsaný v kapitole 5. Vlastní návrh aplikace je popsán v kapitole 6, kde je v podsekci 6.3.1 popsána samotná detekce zranitelnosti typu clickjacking a řešení různých problémů.

Použité technologie při implementaci aplikace spolu s výsledky testování jsou popsány v kapitole 7. Dosažené výsledky spolu s návrhem na budoucí vývoj jsou shrnuty v závěrečné kapitole 8.

Kapitola 2

Hypertext Transfer Protocol (HTTP)

HTTP je protokol pracující na aplikační vrstvě referenčního modelu ISO/OSI a je využíván pro distribuované hypermediální[1] informační systémy. Tato kapitola je napsána na základě normy RFC 2616 [4], vyjma sekce 2.2, která je napsána na základě normy RFC 2109 [7].

2.1 Komunikace

Komunikaci inicializuje klient, vytvořením TCP spojení se serverem (standardně je využíván pro spojení port 80 na straně serveru). Komunikace probíhá mezi klientem a serverem formou zasílání HTTP zpráv. Klient odesílá na server HTTP požadavky, které musí obsahovat všechny potřebné informace, protože protokol je nestavový – server si nepamatuje žádné informace o předešlý požadavcích klienta. Server poté odpovídá zasláním HTTP odpovědi. V případě HTTP/1.0 je poté spojení uzavřeno. V případě HTTP/1.1 lze ustanovit perzistentní spojení pomocí hlavičky *Connection*, což umožnuje také řetězové zasílání několika HTTP požadavků a odpovědí[2].

2.2 Cookies

Cookies slouží pro případné uchování stavu komunikace mezi klientem a serverem. Cookies umožňují uchovávat informace, na základě kterých lze provést například identifikaci uživatele, který tak může zůstat přihlášen u aplikace. Cookies mohou být nastavovány webovou aplikací na straně serveru, pomocí HTTP hlavičky *Set-Cookie*, nebo pomocí javascriptu na straně webové aplikace.

Uchování stavu je zajištěno odesíláním cookies v HTTP požadavku. Cookie obvykle obsahuje název webové stránky, pro kterou je určena, přiřazenou hodnotu a délku platnosti (ve výchozím stavu je cookie smazána při ukončení internetového prohlížeče).

[1] Systém organizace textových, zvukových, grafických dat a videí, který umožnuje přístup k datům na základě asociace.

[2] Klient zasílá HTTP požadavky serveru nezávisle na příchozích odpovědích. Server musí odesílat HTTP odpovědi ve stejném pořadí, v jakém byly přijaty HTTP žádosti.

2.3 Uniform Resource Identifier(URI)

URI (Uniform Resource Identifier) je jednoduše formátovaný řetězec, který slouží k lokalizaci zdrojů na základě jména, lokace (cesty, umístění) a dalších případných informací (např. na základě GET parametrů). V HTTP mohou být URI reprezentována relativní nebo absolutní formou v závislosti na jejich použití. V případě použití absolutního tvaru začíná URI vždy *schématem*, které je ve tvaru `název_scématu:` (v případě HTTP protokolu *http:*).

V případě HTTP se často hovoří o URL, které je podmnožinou URI [3], jehož základní tvar je ve výpisu 2.1. V případě absence portu je defaultně nastaven port 80. Část *query*, začínající otazníkem, obsahuje GET parametry.

```
"http:" "//" host [ ":" port ] [ abs_path [ "?" query ]]
```

Výpis 2.1: Základní tvar URL

2.4 HTTP požadavek

Je odesílán klientem k serveru. Na prvním řádku se vždy vyskytuje použitá metoda, URL a verze HTTP. Povinná je hlavička *Host*, jejímž parametrem je doménové jméno serveru, případně také číslo portu, pokud je vyžadován jiný než standardní port. V případě výskytu těla (zpráva může obsahovat tělo pouze v případě použití metody POST) platí stejná pravidla jako u HTTP odpovědi (viz následující kapitola).

První řádek je oddělen od hlaviček sekvencí *CRLF*, stejně tak je ukončena každá jednotlivá HTTP hlavička. Celá HTTP hlavička (tj. blok hlaviček spolu s prvním řádkem) je ukončena prázdným řádkem, který spolu s ukončením poslední hlavičky tvoří sekvenci *CRLFCRLF*. Stejným způsobem je označen konec zprávy. Název hlavičky je od jeho parametru oddělen pomocí dvojtečky následované mezerou.

Možné metody jsou:

- **GET** – je využívána k získání obsahu objektu (načtení HTML stránky, obrázků, atd...). Případné proměnné jsou obsaženy v URI.
- **POST** – je využívána pouze pro zasílání dat z formulářů, které jsou předány serveru v těle dotazu, takže nejsou vidět v URI. URI může i v tomto případě obsahovat GET parametry, například kvůli navigaci.
- **HEAD** – je využívána pouze k získání hlavičky cílového objektu.
- **PUT** – je využívána k nahrání obsahu souboru na server.
- **DELETE** – požadavek ke smazání cílového objektu na serveru.
- **TRACE** – je používána ke zjištění informací o cíli požadavku na aplikační vrstvě. Například pro zjištění použitých proxy serverů na cestě k cílovému serveru.
- **CONNECT** – je využívána k vytvoření SSL tunelu mezi klientem a serverem skrze proxy server.

První tři metody jsou podporovány také v HTTP/1.0, ostatní pouze v HTTP/1.1. Používány jsou především metody GET a POST, případně CONNECT.

```
GET /html/rfc2616 HTTP/1.1
Host: tools.ietf.org
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: cs,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
```

Výpis 2.2: Hlavička HTTP požadavku pro načtení stránky obsahující RFC 2616. Je použita metoda GET. Obsah odpovědi může být komprimován. Hlavička *Accept* určuje očekávaný MIME typ obsahu.

2.5 HTTP odpověď

HTTP odpověď je odesílána serverem klientovi. První řádek obsahuje použitou verzi HTTP a stavový kód spolu s označením (informující klienta mj. o úspěšnosti vyřízení požadavku). Pro oddělení jendotlivých částí, hlaviček, apod. platí stejná pravidla jako pro HTTP žádost (viz předchozí sekce).

V případě výskytu těla ve zprávě jsou povinné hlavičky *Content-Length*, určující délku těla zprávy, a *Content-Type*, určující MIME typ těla zprávy a případně použitou znakovou sadu. Výjimku tvoří přenos typu *chunked*, oznámený hlavičkou *Transfer-Encoding*, při kterém nemusí být celková velikost obsahu předem známa. Při tomto typu přenosu je obsah přenášen po částech.

```
HTTP/1.1 200 OK
Date: Tue, 14 May 2013 13:47:22 GMT
Server: Apache/2.2.21 (Debian)
Content-Location: rfc2616.html
Vary: negotiate
TCN: choice
Last-Modified: Mon, 13 May 2013 00:11:14 GMT
Etag: "2feb95-808ff-4dc8e5ad1ac80;4dcaddf33a7c0"
Accept-Ranges: bytes
Content-Length: 526591
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Výpis 2.3: Hlavička HTTP odpovědi. Požadavek byl úspěšně zpracován (stavový kód 200). Získána je stránka *rfc2616.html*. Poslední modifikace stránky byla provedena 13. května 2013. Délka obsahu je 526591 Bytů.

Kapitola 3

Bezpečnost webových aplikací

Bezpečnost hraje ve vývoji webových aplikací významnou roli, protože slabá místa v zabezpečení mohou mohou být lehce zneužita i méně znalými uživateli s různým rozsahem škod. Výskyt zranitelností ve webové aplikaci není pouze důsledkem určité neznalosti vývojáře, ale také jeho nepozorností, která se může při ošetřování mnoha vstupů a výstupů snadno vyskytnout. Do určité míry lze tento problém řešit například použitím vhodného frameworku, který se o ošetření vstupů a výstupů postará automaticky (např. populární framework Nette[1]).

Další možností je penetrační testování, kterému je věnována sekce 3.4. Ve zmíněné sekci jsou také uvedeny a popsány dva známé penetrační nástroje – Paros (viz 3.4.1) a BurpSuite (viz 3.4.2). Existuje celá řada penetračních nástrojů, ale na těchto dvou příkladech jsou nejlépe vidět výhody a nevýhody plně automatizovaných testů, kdy uživatel nemá šanci jakkoliv ovlivnit jednotlivé testy, a plně nastavitelných testů, kdy musí uživatel manuálně nastavit všechny parametry testů.

Zranitelnosti mohou ovšem pocházet také z internetových prohlížečů, které představují hrozbu především pro samotné uživatele. Pro větší bezpečnost jsou tedy sestavovány bezpečnostní politiky, které se snaží zvýšit bezpečnost uživatelů webových aplikací (viz sekce 3.2 a 3.3).

Následující sekce se zabývá zranitelností a útokem XSS a byla sepsána na základě knihy Romana Kümmela [8]. XSS je z pohledu této práce podstatný, protože je velmi rozšířený a s jeho pomocí lze zneškodnit zabezpečení proti clickjackingu, které je tvořeno *framebusterem*.

3.1 Cross-site scripting (XSS)

Při XSS útoku se snaží útočník vložit svůj kód do obsahu webové stránky, který je vykonán prohlížečem na straně uživatele. Tento útok je tedy veden především proti samotným uživatelům, nikoli tedy přímo proti webové aplikaci. Zranitelnost spočívá především v neošetřeném výstupu, jehož obsah pochází od uživatele (např. návštěvní kniha). Základní ošetření lze provést záměnou nebezpečných znaků za jejich HTML entity (viz tabulka 3.1). V PHP existuje k tomuto účelu funkce `htmlcpecialchars()`. Přes poměrně jednoduchý způsob ošetření, který lze aplikovat pokud není uživatelům umožněno vkládání určitých HTML tagů, se jedná o jednu z nejrozšířenějších zranitelností webových aplikací.

XSS útoky lze rozlišit do tří kategorií:

[1] http://nette.org/cs/

Speciální znak	HTML entita
"	"
’	'
<	<
>	>
&	&

Tabulka 3.1: Speciální znaky a jejich HTML entity.

- **Stored XSS** – neboli také trvalé či perzistentní XSS
- **Reflected XSS** – neboli také non-perzistentní XSS
- **DOM-based XSS**

V případě perzistentního XSS je injektovaný kód uložen do datového uložiště aplikace, odkud je poté automaticky načítán při každém zobrazení odpovídající stránky. Jde o nejnebezpečnější formu XSS. V tomto případě postačí pro demonstraci neškodný javascriptový kód, který pouze otevře na stránce dialogové okno s textem */XSS/*, a stránka s návštěvní knihou. Na obrázku 3.1 lze vidět stránku s již vyplněným formulářem. Po odeslání dat bude proveden zápis do datového úložiště a při každém dalším zobrazení stránky uvidí uživatel zmiňované dialogové okno (viz obr. 3.2).

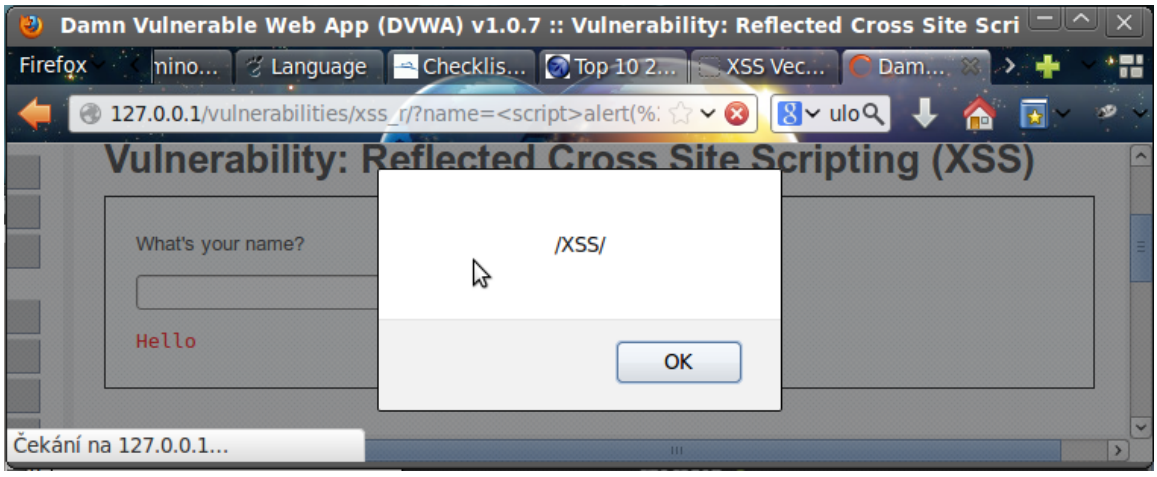
Obrázek 3.1: Návštěvní kniha se zranitelností umožňující perzistentní XSS s již vyplněným formulářem. V textovém poli *Message* je vepsán injektovaný kód.

U non-perzistentního XSS je kód injektován do obsahu webové stránky pouze pro daný HTTP požadavek a není nikde trvale uložen. Jde o nejčastěji se vyskytující formu XSS. Do určité míry je tento typ útoku zachytitelný vestavěnými XSS filtry určitých prohlížečů[2]. Na

[2]XSS filtry jsou vestavěny v prohlížečích IE od verze 8, Chrome a Firefox s přídavným modulem NoScript.

Obrázek 3.2: Na stránce byl úspěšně proveden perzistentní XSS útok. Při načítání stránky je automaticky z datového úložiště nahrán injektovaný kód.

obrázku 3.4 je vložen injektovaný kód do obsahu GET parametru `name`.



Obrázek 3.3: Ukázka non-perzistentního XSS. V adresním řádku lze vidět část URI s injektovaným kódem (kód je totožný jako v příkladě perzistentního XSS).

DOM-based XSS je velice podobný non-perzistentnímu XSS. Rozdíl je ve způsobu vkládání injektovaného kódu. Zatímco u non-perzistentního XSS je veškerý obsah vložen již na straně serveru, v případě DOM-based XSS dochází ke vložení hodnot z parametrů na straně webového prohlížeče javascriptem. Příklad takové zranitelnosti může představovat HTML stránka 3.1, na které je vypsán pomocí javascriptu obsah GET proměnné `name`. Útočník by navíc mohl využít toho, že v odesílaném HTTP požadavku není zahrnuta část

URI nacházející se za mřížkou # včetne mřížky. V prohlížeči je však tato část přístupná. Postup injektáže je zde shodný jako u předešlého případu. Kód je nyní pouze umístněn za zmíněnou mřížkou, které předchází text „Karel“.

```
<!DOCTYPE html>
<html>
<head><title>DOM-based XSS</title></head>
<body>
 <p>Obsah GET parametru <b>name</b> je:
   <script>
     document.write(unescape(document.location.href
           .match("[&?]name=[^&]*").toString().match("[^=]*$")));
   </script>
 </p>
</body>
</html>
```

Výpis 3.1: Stránka zranitelná proti DOM-based XSS útoku.

Obrázek 3.4: Ukázka DOM-based XSS útoku.

3.2 Same Origin Policy (SOP)

Javascript může sloužit vývojářům i útočníkům jako velmi mocný nástroj. V případě SOP se jedná o jednu z nejdůležitějších bezpečnostních norem, které byly v rámci webových prohlížečů vydány. Tato norma zavádí omezení v přístupu k datům či obsahu webových stránek z různých domén. To znamená, že vývojáři mohou manipulovat pomocí javascriptu pouze s vlastními stránkami a nemají přístup k obsahu stránek z jiných domén, které by byly například zanořeny ve stránce pomocí HTML tagu `<iframe>`. [8]

3.3 Content Security Policy (CSP)

Jak již bylo zmíněno v sekci 3.1, XSS je jednou z nejrozšířenějších zranitelností webových aplikací. Vestavěné XSS filtry poskytují pouze určitou ochranu proti XSS útokům, které nejsou perzistentní. V reakci na stávající hrozbu XSS útoků vznikla bezpečnostní politika CSP.

Myšlenka CSP je založena na těsnější spolupráci webové aplikace a prohlížeče za účelem specifikace legitimních zdrojů dat. Prvním předpokladem pro funkčnost CSP je podpora ze strany prohlížeče[3]. Webovou aplikací poté musí být explicitně sděleno, že má být v rámci stránky uplatněna zmíněná bezpečnostní politika. Pomocí direktiv lze následně určit legitimní zdroje pro skripty, obrázky, atd. Pro vyšší účinnost jsou defaultně zavedena také určitá omezení. Patří zde například zákaz přímého vkládání skriptu pomocí HTML tagu `<script>` nebo definování ovladačů událostí přímo u jednotlivých HTML elementů. [1, 8]

3.4 Penetrační testování

Penetrační testování je technika využívaná k nalezení bezpečnostních děr v systému či aplikaci, známá také jako etický hacking. V případě webových aplikací se tester se snaží naleznout pokud možno co nejvíce zranitelností v zabezpečení aplikace. Testování je prováděno nejčastěji formou nedestruktivního útoku[4], dle jehož úspěšnosti je či není detekována určitá zranitelnost v aplikaci.

Proces testování významně usnadňují nástroje, které dokážou testovat webovou aplikaci i automatizovaně na určité zranitelnosti. Často je tímto způsobem prováděno základní testování, které může odhalit nejběžnější zranitelnosti, jako je například XSS, SQL injection, aj. Bohužel automatizovaná detekce není vždy nejspolehlivější a nenalezení žádné zranitelnosti tak nemusí hned znamenat její absenci v aplikaci.

3.4.1 Paros

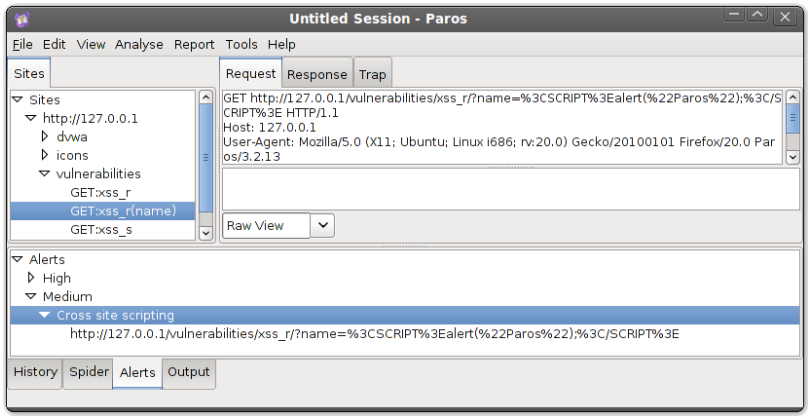
Jedním z nejznámějších nástrojů určených k testování zabezpečení webových aplikací je Paros[5] spadající do kategorie proxy serverů. Nástroj je zdarma a je napsán v Javě. Umožnuje zachytávání HTTP(S) komunikace mezi klientem a serverem, automatizované mapování webu a také automatizované testování webových aplikací na často se vyskytující zranitelnosti.

Mezi výhody zde patří jednoduché intuitivní prostředí, umožňující automatizované testování webových aplikací i méně zkušeným uživatelům, platformová nezávislost a otevřenost zdrojových kódů. Nástroj si tak mohou uživatelé dále upravovat a rozšiřovat dle svých potřeb. Nevýhodou je chybějící možnost výběru libovolné množiny stránek k testovaní. Dále nelze v uživatelském prostředí nijak nakonfigurovat jednotlivé typy testů. Částečně je tato nevýhoda kompenzována možností vytvoření vlastního HTTP požadavku nebo možností zásahu do kódu aplikace.

[3] Určité experimentální implementace byly provedeny v prohlížečích Firefox a Chrome již před vydáním standardu. V prohlížeči Internet Explorer byla zavedena částečná podpora až v aktuální verzi 10. Aktuální informace o podpoře CSP webovými prohlížeči jsou k nalezení na http://caniuse.com/contentsecuritypolicy.

[4] Destruktivním útokem je zde myšlen útok, který by mohl napáchat v aplikaci či databázi významné škody, které by mohly poškodit klienta.

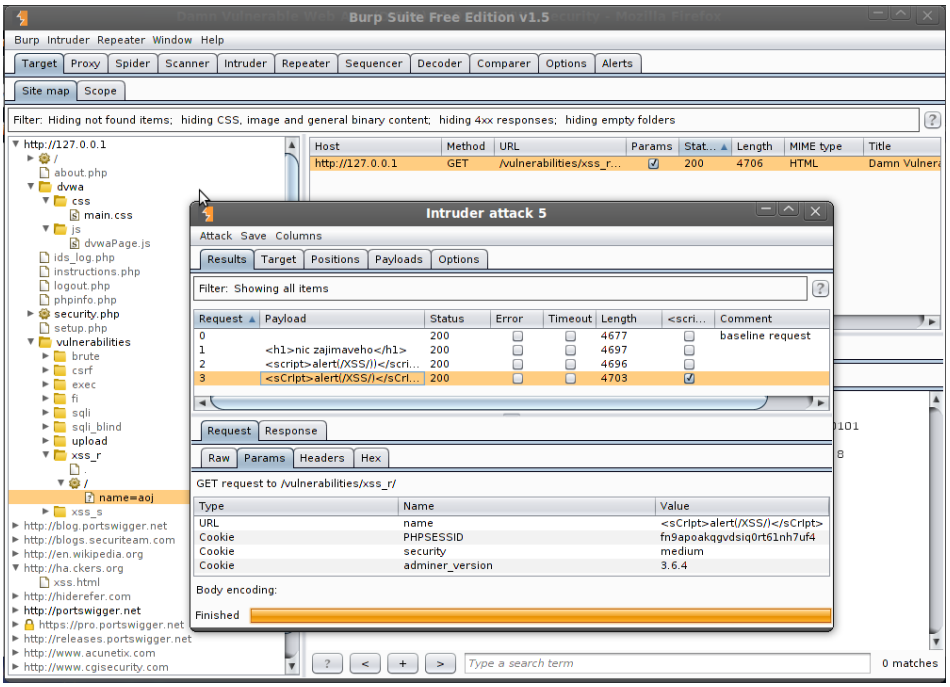
[5] http://www.parosproxy.org/



Obrázek 3.5: Automatizovaná detekce zranitelnosti XSS v nástroji Paros.

3.4.2 BurpSuite

BurpSuite je profesionální nástroj k testování zabezpečení webových aplikací. Obdobně jako Paros spadá do kategorie proxy serverů a je implementován v Javě. Poskytuje však širokou škálu funkcí, které výrazně usnadňují a zefektivňují vyhledávání bezpečnostních děr. Bohužel je tento nástroj placený a v dostupné demoverzi chybí například automatizované testování a nelze ukládat jednotlivá sezení. Stále je však tato aplikace použitelná k manuálnímu vyhledávání různých zranitelností. Nevýhodou může být zpočátku obtížnější orientace. Na internetu lze však najít různé tutoriály, které poměrně rychle zasvětí uživatele do základních funkcí aplikace.



Obrázek 3.6: Manuální detekce zranitelnosti XSS v demoverzi nástroje BurpSuite.

Kapitola 4

Clickjacking

Clickjacking je webový útok, který využívá zranitelností internetových prohlížečů. Některé jeho formy mohou být závislé na daném prohlížeči a některé jsou naopak nezávislé. Rovněž jej lze označit za platformově nezávislý. S clickjacking útoky se lze dnes nejčastěji setkat na sociálních sítích (např. Facebook, Twitter). Důvody, proč jsou sociální sítě u útočníků tak oblíbené, jsou především dva. Prvním z nich je vysoký počet uživatelů těchto sítí (samotný Facebook má k dnešnímu dni více než 950 miliónů aktivních účtů po celém světě[1]). Druhým důvodem je zde časté využívání trvalého přihlášení uživatelů v aplikaci. Jak vyplývá z předchozího tvrzení, představuje clickjacking pro oběť největší nebezpečí především v případě, kdy je útok vytvořen pro aplikaci, ke které je oběť přihlášena nebo je útočník schopen oběť k této aplikaci přihlásit pod jejím účtem. V takových případech je poté útočník při znalosti zdrojového kódu stránky schopen nechat oběť nevědomky provádět v aplikaci úkony, které potřebuje [5].

Tento útok může být využit k různým účelům s různými škodami. Tím nejméně nebezpečným může být například ovlivnění výsledků ankety. Přestože k tomuto účelu je možné použít například útok CSRF, clickjaking lze využít často i na stránkách, které jsou proti útoku CSRF již zabezpečeny. Clickjacking může být využit i k provedení dalších útoků, jako například XSS, kdy clickjacking je využit k prolomení ochrany, která je tvořena jen XSS filtry některých prohlížečů.

4.1 Analýza útoku

Při clickjacking útoku podstrčí útočník oběti podvodnou internetovou stránku, se kterou oběť interaguje, přičemž jednotlivá kliknutí útočník využívá k provádění úkonů na jiné stránce, aniž by o tom oběť věděla. Tím útočník docílí toho, že jednotlivé akce jsou prováděny s právy a identitou oběti a nemusí se tedy sám za oběť vydávat. [11]

Mylně bývá clickjacking někdy zaměňován za CSRF. Rozdíl proti CSRF útoku spočívá v provádění jednotlivých akcí přímo na napadené stránce, zatímco při CSRF lze pouze vytvářet požadavky na podvodné stránce, které jsou následně odesílány napadené stránce.

Podle Roberta Hansena má clickjacking útok mnoho podob [5]. Může využívat javascriptu, XSS, CSRF, ale také nemusí. Nicméně, Michal Zalewski definoval několik způsobů, jak může útočník principiálně vytvořit podvodnou stránku a vést tak svůj útok [13]:

- Útočník vloží cílovou stránku pomocí HTML tagu `<iframe>`, kterou umístí nad viditelný obsah a pomocí CSS ji udělá plně průhlednou.

[1] http://checkfacebook.com

- Útočník vloží cílovou stránku pomocí HTML tagu `<iframe>` a následně ji překryje několika elementy tak, aby zůstaly viditelné pouze ty části stránky, na které má oběť kliknout.
- Útočník vloží cílovou stránku pomocí HTML tagu `<iframe>`, který ořeže na velikost oblasti, do které má uživatel kliknout. Poté nastaví pohled v zobrazované stránce tak, aby tato oblast vyplňovala viditelný výřez, a odstraní posuvníky. Celý výřez pak vhodně umístí do zobrazované podvodné stránky.
- Útočník vloží stránku do samostatného nového okna nebo rámce a celou ji skryje. Poté ji vhodným načasováním převede do popředí tak, aby oběť již nestihla na změnu zareagovat.

Asi nejpoužívanější variantou je první ze zmíněných způsobů. Díky tomu, že není cílová stránka viděna po celou dobu interakce, lze relativně jednoduše vytvořit komplexní útoky, které vyžadují i více akcí od oběti. Například v prohlížečích, které podporují funkci *drag&drop*, lze postupně vyplnit například email a poté jej odeslat nebo naopak lze extrahovat informace z cílové stránky [11, 12].

Naproti tomu poslední jmenovaný způsob je používán nejméně, protože je také nejméně spolehlivý. Nicméně i zde existují techniky, které mohou úspěšnost útoku zvýšit. Jedna z těchto těchto technik byla pojmenována *Double-clickjacking* [6]. Jak již tento název napovídá, je cílem útočníka přesvědčit oběť, aby na určitý element použila dvojklik. Mezi prvním a druhým kliknutím se přenese do popředí dříve cílová stránka a oběť již většinou nestihne včas na tuto změnu zareagovat.

4.2 Zabezpečení

Nehledě na množství důsledků, které může ignorace tohoto problému mít, je zabezpečení aplikace ve většině případech relativně jednoduché. Celá obrana je založena na zákazu zobrazení standardního obsahu stránky v případech, kdy je zanořena uvnitř jiné stránky pomocí HTML tagu `<iframe>`. Toho lze docílit několika způsoby, které jsou v následujících sekcích podrobně rozebrány. Je nutné podotknout, že samostatné zabezpečení proti clickjackingu způsoby, které jsou zde prezentovány, není naprosto spolehlivé, pokud je stránka náchylná proti dalším útokům, jako je například XSS[2]. V takových případech by mohl útočník v budoucnu znehodnotit zabezpečení webu, který by se tak mohl stát zranitelným proti clickjacking útoku.

Sekce 4.2.1 a 4.2.3 (včetně převzatých algoritmů) byly sepsány na základě publikace [11].

4.2.1 Zabezpečení javascriptem

Ochrana spočívá ve vytvoření krátkého javascriptového kódu na stránce, která má být zabezpečena. Ten nejprve ověří, zda není stránka zanořena uvnitř jiné stránky a v případě zjištění, že se tak stalo, vykoná protiakci, která spočívá v pokusu o přesměrování na vlastní stránku nebo změně obsahu stránky. Tento kód bývá často v praxi označován jako *framebuster*, *framekiller* či *framebreaker*. V nejjednodušším případě může kód vypadat takto:

[2] Viz sekce 3.1.

```
<script>
  if(top !== self) {
    top.location = self.location;
  }
<script>
```

Výpis 4.1: Prolomitelný framebuster

Nicméně tato ochrana je nedostatečná. Existuje hned několik způsobů, jak toto zabezpečení obejít, které jsou poměrně známé a snadno dohledatelné. Přitom zranitelnost tohoto kódu spočívá především v závislosti na úspěšném provedení přesměrování (oba zmíněné problémy budou rozebrány níže). Bohužel, vývojáři webových aplikací (dále jen vývojáři) se při psaní tohoto druhu zabezpečení dopouštějí také dalších zásadních chyb.

Existuje několik způsobů jak zjistit zda je stránka zanořená nebo jak provést samotné přesměrování. Hrubé chyby se vývojáři dopouští při práci s objektem `parent`. Zatímco v podmínce testování pomocí tohoto objektu nevadí, pokud není počítáno s možností udělování výjimek spřáteleným webům (viz sekce 4.2.3), u přesměrování je situace opačná. Pomocí objektu `parent` se vývojář pokouší o přesměrování rodičovského prvku. V případě jednoduchého zanoření, kdy je rodič zároveň hlavní stránkou, na kterou se oběť dostala, je vše v pořádku. Ale v případě, že je takto zabezpečená stránka zanořena uvnitř jiné již zanořené stránky, bude tento požadavek o přesměrování zamítnut v rámci porušení pravidel SOP, kterými se dnes řídí všechny populární internetové prohlížeče. Nutno poznamenat, že bez této bezpečnostní politiky by byl internet mnohem nebezpečnější a tomuto problému se lze snadno vyhnout například využitím objektu `top`.

Dalším problémem je testování zanoření stránky pomocí objektu `location`. V případě prohlížečů Microsoft Internet Explorer (dále jen IE) verze 7 a Safari verze 4.0.4 bylo možné tento objekt redefinovat. Čímž lze ovlivnit výsledek tohoto testování a v určitých případech také znemožnit přesměrování na jinou stránku. Experimentálně jsem ověřil, že tato chyba je opravena v IE od verze 9. Prokazatelně tuto chybu obsahuje i prohlížeč Safari verze 5.1. Na novějších verzích tohoto prohlížeče nebyl tento test proveden.

Jak již bylo zmíněno, nevýhodou ochrany v ukázce kódu 4.1, je závislost na úspěšném provedení přesměrování. Problém tedy může představovat situace, kdy je na stránce zakázáno vykonávání javascriptového kódu. Tuto situaci může navodit jak samotný uživatel (oběť), tak útočník, který může nastavit načtení stránky ve speciálním módu. Ať již využije atributu `security=restricted` v IE, `designMode` v Mozille Firefox (dále jen Firefox) nebo `sandbox`, který přináší HTML5, bude tato ochrana ve většině prohlížečů neučinná. Navíc dnes útočník může zneužít XSS filtry zabudované v příhlížečích IE od verze 8 a Google Chrome (dále jen Chrome), díky kterým může deaktivovat provádění pouze nevyhovujících částí kódu. Tím se riziko rozšiřuje i na stránky, které pro svou funkcionalitu javascript přímo vyžadují.

Další účinnou metodou pro obcházení tohoto zabezpečení je zrušení přesměrování, čehož lze dosáhnout dvěmi způsoby. První z nich vyžaduje spolupráci oběti, která je při pokusu o přesměrování dotázána, zda chce opustit aktuální (podvodnou) stránku (ukázka kódu 4.2). V tomto případě závisí vše na oběti. Bystrý a podezřívavý uživatel může pojmout podezření a stránku úplně opustit. Ale je pravděpodobné, že uživatel přeruší přesměrování, protože si chce prohlédnout obhsah aktuální stránky a o žádné přesměrování nežádal.

```
<script>
window.onbeforeunload = function() {
  return "Opravdu chcete opustit tento web?";
}
```

```
</script>
```

Výpis 4.2: Přerušení přesměrování na základě rozhodnutí oběti

Aby se útočník vyhnul riziku, že vzbudí u oběti nějaké podezření, využije nejspíše druhé metody, u které nepotřebuje spolupráci od oběti (ukázka kódu 4.3). Funkčnost této metody spočívá v reakci prohlížeče na obdržení HTTP odpovědi se stavovým kódem *204*, který prohlížeči oznamuje, že na dané URL adrese se nenachází žádný obsah. Útočník, před samotným přesměrováním, odešle vlastní žádost o přesměrování na URL, z které zpětně obdrží výše zmíněnou odpověď. Prohlížeč v tomto případě přesměrování na tuto adresu neprovede a navíc odstraní ostatní již čekající požadavky, mezi kterými je i původní požadavek o přesměrování na vlastní stránku. Tato metoda je funkční na všech populárních prohlížečích[3].

```
<script>
var kill_bust = 0
window.onbeforeunload = function() {kill_bust++}
setInterval(function() {
  if (kill_bust > 0) {
    kill_bust -=2;
    window.top.location = 'http://no-content-204.com'
  }
},1);
</script>
```

Výpis 4.3: Přerušení přesměrování bez vědomí oběti

Z těchto důvodů je dnes doporučován mírně upravený bezpečnostní kód, který kombinuje zákaz vkládání stránky na cizím webu s pokusem o přesměrování a zobrazení standardního obsahu pouze v případě, že není zjištěno zanoření stránky. Nejjednodušším způsobem jak zabránit zobrazení obsahu je nastavení CSS atributu `display` u elementu `body`.

```
<style> body { display : none;} </style>
...

<script>
if (self === top) {
  var theBody = document.getElementsByTagName('body')[0];
  theBody.style.display = "block";
} else {
  top.location = self.location;
}
</script>
```

Výpis 4.4: Bezpečná ochrana proti clickjackingu

4.2.2 X-Frame-Options

Další možností obrany proti clickjackingu je využití HTTP hlavičky `X-Frame-Options`, která pochází od firmy Microsoft a je dnes již podporována všemi populárními prohlížeči. Tato hlavička obsahuje jediný parametr, který může nabývat standardně hodnoty DENY nebo SAMEORIGIN. V prvním případě bude zakázáno zanoření stránky úplně, ve druhém bude zanoření povoleno v rámci stejné domény. [9]

[3] Naposled experimentálně ověřeno na aktuálních verzích internetových prohlížečů ke dni 4. května 2012

Z toho vyplývá, že tato forma obrany není vhodná v případě, kdy je vyžadováno udělování výjimek, protože zde chybí možnost uplatnit whitelist. V implementaci některých prohlížečů sice existuje možnost využití třetího nestandardního parametru `ALLOW-FROM`, kde lze zadat jednu vybranou doménu, která bude mít udělenu výjimku, ale v případě potřeby udělování více výjimek je tato volba stále nedostačující. Navíc není podporována všemi populárními prohlížeči. Částečně by šlo problém vyřešit pomocí PHP skriptu, kdy by mohla být provedena kontrola na základě hlavičky `Referer` a v případě spřáteleného webu by hlavička `X-Frame-Options` nebyla odeslána. Ale nelze se na toto řešení spolehnout, protože uživatelé mohou mít odesílání hlavičky `Referer` zakázáno a v některých případech se hlavička neodesílá vůbec, čehož dokáže útočník rovněž využít. [9]

Další problém zde tvoří proxy servery, které mohou hlavičku zahodit a prohlížeč tak nebude vědět, že má zobrazení stránky zabránit. V prohlížečích Chrome a Safari lze předat hlavičku také pomocí atributu `http-equiv` v HTML tagu `<meta>`, ale jedná se o nestandardní řešení, které je účinné pouze pro část uživatelů internetu. Proto je doporučováno použití tohoto řešení v kombinaci s javascriptovým kódem.

4.2.3 Povolení výjimek

V některých případech se mohou vyskytnout požadavky na povolení vkládání stránky na spřátelené weby. Jediné spolehlivé řešení dnes představuje upravení ochrany pomocí javascriptového kódu, který byl popsán v sekci 4.2.1. Úprava spočívá ve změně testovací podmínky, kdy je otestována URL hlavní stránky[4] například pomocí regulárního výrazu. Zde se vývojáři mohou dopustit několika chyb.

Zatímco v případě zabezpečení bez udělování výjimek byla většina problémů spojena s protiakcí, která se měla vykonat, nyní se problémy přesouvají stejnou měrou na testovací podmínku, na jejímž základě má být protiakce vykonána. V prvním případě je podstatný zdroj informace, který bude využit ke zjištění URL hlavní stránky. I v tomto případě je vhodné zůstat u objektu `top`, z něhož lze spolehlivě získat URL hlavní stránky. V případě získání URL z objektu `parent.location` nebo `document.referrer` bude zabezpečení stránky závislé i na zabezpečení spřáteleného webu. Pokud bude povoleno vkládání stránky na spřáteleném webu, který již nebude proti clickjackingu dobře zabezpečen, může nastat situace, kdy útočník na své stránce zanoří takto spřátelený web a skrze něj poté provede útok na cílovou stránku. Zatímco v případě, kdy je zjištěna URL hlavní stránky z objektu `top.location`, zůstává ochrana vlastní stránky nezávislá na zabezpečení ostatních webů.

Další častou chybou, které se při kontrole domény vývojáři dopouští, je špatně sestavený regulární výraz. Nestačí sestavit regulární výraz, který bude vyhledávat v URL klíčová slova (např.: `/.*(seznam|firma).*/`). V tomto případě by test skončil kladným výsledkem také v případě, kdy by tato klíčová slova byla obsažena v URL (např.: `http://www.seznamobleceni.cz` nebo `http://www.podvodnik.cz?key=firma`). Je proto nutné být při sestavení regulárního výrazu důslednější (např.: `/^https?:\/\/(www\.)?(seznam\.cz|firma.net)\/.*/`).

Spolehlivé řešení tohoto problému by v budoucnu mohla představovat bezpečnostní politika CSP. Konkrétně direktiva `frame-ancestors`, pomocí které by bylo možné nastavit seznam webů, které by měly povoleno zanoření dané stránky. Bohužel tato direktiva nebyla zahrnuta v nedávno sestaveném prvním standardu CSP a tak lze jen čekat, zda bude do

[4] Zde je tím myšlena stránka, jejíž URL adresa je zobrazena v adresním řádku prohlížeče.

standardu zahrnuta někdy v budoucnu. Nyní je tato direktiva nestandardní a nelze se spoléhat na její podporu v prohlížečích. [11, 1]

4.2.4 Možnosti uživatele

Zranitelnost typu clickjacking je ze strany vývojářů často podceňována [2] a uživatel by se měl proto zajímat, jak by se mohl proti možným útokům bránit sám. Z běžně dostupných a známých prostředků dnes poskytuje určitou ochranu rozšíření *NoScript*, které je bohužel omezené pouze na prohlížeč Firefox, kvůli závislosti na renderovacím jádru *Gecko*. Toto rozšíření obsahuje mj. modul *ClearClick* (viz sekce 4.3.1) na jehož vývoji se podíleli také Robert Hansen a Jeremiah Grossman [10]. Některé uživatele může odradit počáteční fáze využívání tohoto rozšíření, kdy je potřeba NoScript „naučit“, které stránky jsou důvěryhodné a kde může být spouštěn javascript. Po počáteční fázi se stává jeho používání jednoduché a nijak výrazně nepřekáží.

Jediným dalším běžně dostupným produktem, který jsem nalezl, je *Comitary Web Protector*[5], který v základní verzi, která je zdarma, nabízí mj. ochranu před clickjacking útoky. Podporovány jsou prohlížeče Firefox, IE a Chrome. Jeho nevýhodou je závislost na operačním systému Microsoft Windows.

Nicméně pořád platí, že nejlepší obranou proti clickjackingu je obecná obezřetnost. S clickjackingem se lze často setkat například na stránkách, které mají značně nedotažený design a obsah. Často se s nimi lze setkat například na sociální síti Facebook, kde jsou uživatelé lákáni zpravidla na titulky hlásající například: „To musíte vidět! Celebrita v rouše evině! Pro zobrazení dejte Like!“ apod. Existují ovšem i mnohem propracovanější weby, u kterých by uživatel opravdu nemusel pojmout žádné podezření. I v tomto případě uživatel není bezmocný. Uživatel by neměl ukládat v prohlížečích svá hesla a využívat předvyplňování přihlašovacích formulářů. Dále by se měl odhlašovat z webových aplikací poté, co je přestane užívat. Pro ještě vyšší bezpečnost lze využívat dvou internetových prohlížečů, přičemž jeden by byl využíván například k pohybu v internetovém bankovnictví, práci a dalších citlivějších činnostech, a druhý by byl využíván například k pohybu po zábavných stránkách a stránkách s méně důvěrhodným obsahem. Riziko, že by byly uživateli napáchány významější škody, takto významně klesá.

4.3 Detekce

Manuální detekci zranitelnosti typu clickjacking zvládne poměrně snadno každý uživatel znalý tohoto útoku. Stačí vytvořit HTML stránku, která bude obsahovat plovoucí rám se stránkou, která má být zkontrolována. Podle používaného prohlížeče se nastaví rovněž atribut, který zamezí spouštění javascriptu u testované stránky. V případě, že uživatel spatří v plovoucím rámu cílovou stránku se standardním obsahem, obsahuje daný web zranitelnost typu clickjacking.

```
<html>
  <head><title>Clickjacking Test</title></head>
  <body>
    <iframe src="http://testovana-stranka.cz"
        style="width:400px;height:400px;" sandbox="">
    </iframe>
  </body>
```

[5] Dostupný na http://www.comitari.com/Web_Protector.

```
</html>
```

Výpis 4.5: Ukázka možného HTML kódu stránky k testování zranitelnosti typu clickjacking

V případě automatizované detekce je situace o něco složitější. Otázkou automatizované detekce clickjackingu se zabýval tým specialistů na internetovou bezpečnost, který vedl Marco Balduzzi. Výstupem jejich práce je nástroj, který se skládá z detekční a testovací jednotky.

Detekční jednotka slouží k detekci clickjackingu. Je složena z modulů ClearClick, který již byl vyvinut dříve pro rozšíření NoScript, a ClickIDS, který byl vyvinut tímto týmem v rámci tohoto řešení (oba moduly jsou popsány níže). Při propojení obou těchto modulů dosáhli uspokojivé detekce clickjackingu s výrazným snížením falešně pozitivních nálezů[6], které generovaly oba moduly odděleně (viz tabulka 4.1). [2]

Testovací jednotka slouží k simulaci uživatelských kliknutí na testované stránce. Nejprve je prohlížeč vyzván k načtení určité internetové stránky. Po načtení této stránky je kurzor myši postupně přemisťován na souřadnice všech vykreslených klikatelných elementů[7], které jsou posléze pokryty kliky[8]. [2]

	Celkem	**True Positives**	**BorderLines**	**False Positives**
ClickIDS	137	2	5	130
NoScript	535	2	31	502
Oba	6	2	0	4

Tabulka 4.1: Srovnání výsledků jednotlivých modulů a jejich propojení. [2]

Další možností, která již je zaměřena na detekci zranitelnosti webu proti útokům typu clickjacking, je vyhledávání ochranných prvků[9], které znemožňují úspěšné provedení útoku. Web je v takovém případě vyhodnocen jako zranitelný, pokud

- není nalezen žadný z těchto bezpečnostních prvků
- nebo jsou nalezeny bezpečnostní prvky, ale ani jeden z nich není považován za dostatečně spolehlivý.

4.3.1 ClearClick

Modul ClearClick je součástí rozšíření NoScript, vytvářené pro prohlížeč Firefox, a poskytuje určitou ochranu proti clickjacking útokům. Vychází z předpokladu, že při clickjackingu se snaží útočník skrýt pravou podstatu toho, na co uživatel kliká. Kdykoli uživatel klikne na stránku vloženou v rámci nebo na *plugin objekt*[10], vytvoří ClearClick v oblasti kliknutí obraz daného prvku bez průhlednosti a prvků, které jej překrývají. Výsledný obraz je porovnán s výstupem, který je prezentován uživateli a v případě rozdílnosti obou obrazů modul detekuje možný pokus o clickjacking. Následně uživatel spatří varovné dialogové okno, kde je upozorněn, že by se mohl stát obětí tohoto útoku. K tomuto hlášení je také připojen originální obraz prvku, na který bylo kliknuto, a uživatel se může sám rozhodnout, zda

[6] Běžně označováno jako False Positive (FP).
[7] Popis klikatelných elementů se nachází v sekci 4.3.2.
[8] Kliknutím je zde myšlena simulace kliknutí uživatele na levé tlačítko myši.
[9] Framekiller kód nebo HTTP hlavička X-Frames-Options. Viz sekce 4.2.
[10] Například flash aplikace.

se jedná o útok či falešné hlášení. Nevýhodou tohoto řešení je vysoké množství falešných hlášení. [10, 2]

4.3.2 ClickIDS

ClickIDS byl vyvinut jako součást *detekční jednotky* týmem specialistů pod vedením M. Balduzziho. ClickIDS sleduje pozici kurzoru myši a v případě, že se kurzor objeví na souřadnicích, které jsou obsazeny dvěmi a více klikatelnými prvky z různých stránek, je náhlášeno podezřelé chování. Mezi klikatelné prvky jsou zde počítány odkazy (HTML tag `<a>`), tlačítka, vstupy formulářů (textová pole, checkboxy, radio tlačítka a rolovací seznamy) a dodatečně také obsah Adobe Flash v HTML tagu `<embed>`. Nevýhodou tohoto řešení je nemožnost detekce útoku u částečně zakryté stránky[11] tak, jako to dokáže modul ClearClick. Výhodou tohoto řešení je naopak nižší počet falešných hlášení. [2]

[11] Druhý typ útoku, který byl popsán v sekci 4.1.

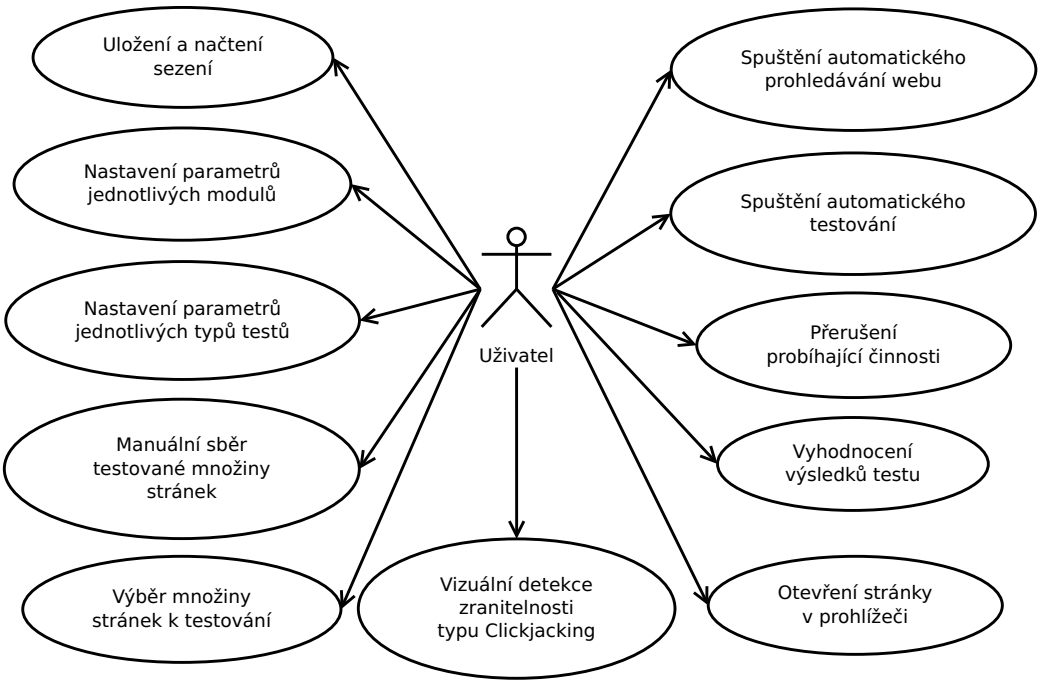
Kapitola 5

Analýza

Hlavním cílem této práce bylo vytvoření aplikace, která by sloužila k detekci zranitelnosti typu clickjacking ve webové aplikaci. Kromě tohoto cíle jsem s přihlédnutím k dalším nástrojům stanovil několik vlastní požadavků, kterých bych chtěl dosáhnout.

5.1 Vlastní požadavky

Aplikace by měla být schopná zaznamenat internetové stránky navštívené uživatelem s použitím internetového prohlížeče a to jak v případě využití samotného HTTP protokolu, tak v případě komunikace skrze SSL tunel. Stejně tak by měla být schopná automatizovaného mapování webu. Pro testování by měl mít uživatel možnost vybrat libovolnou množinu stránek z různých domén z databáze skládající se ze stránek navštívených uživatelem v prohlížeči nebo získaných automatizovaným mapováním.



Obrázek 5.1: Diagram případů užití aplikace WebSerpent.

Uživatel by měl mít možnost uložení a načtení relace, kde by byla uložena databáze stránek spolu s výsledky provedených testů. Kromě detekce samotného clickjackingu by měla být aplikace rozšiřitelná o detekci dalších typů zranitelností, mj. z důvodu potenciální hrozby narušení zabezpečení proti clickjackingu využitím jiných zranitelností (viz následující sekce).

Aplikace by měla mít jednoduché a intuitivní grafické uživatelské rozhraní a podávat srozumitelné výsledky jednotlivých testů, na jejichž základě by byl uživatel schopen posoudit, zda se jedná o zabezpečený či nezabezpečený web. Měla by poskytovat automatizované testování, které by bylo jednoduché jako u nástroje Paros[1], ale zároveň poskytovalo možnost určité parametrizace jednotlivých testů, kterou by uživatel mohl nastavit testování dle vlastních potřeb a zvýšit tak spolehlivost výsledků testů. Zároveň by měla být uživateli poskytnuta možnost manuální detekce clickjackingu[2], kterou by mohl využít k rychlému ověření správnosti výsledků u sporných případů.

Výsledná aplikace by měla být multiplatformní, aby byla využitelná minimálně v operačních systémech Microsoft Windows a Linux.

5.2 Detekce ostatních zranitelností

Přestože může být web na první pohled zabezpečený proti clickjacking útokům, může být toto zabezpečení narušeno skrze jiné bezpečnostní díry pomocí jiných webových útoků. Nejpravděpodobnější zranitelností, která se na webu může vyskytovat, je zranitelnost typu XSS[3], jejíž zneužití může být jednoduché a může mj. vést k narušení ochrany, která je tvořena *framebasterem*. Pro její rozšířenost je využití této zranitelnosti pro narušení zebezpečení proti clickjackingu nejpravděpodobnější. Proto by měla aplikace zahrnovat testy k detekci této zranitelnosti.

Existují také další zranitelnosti, které lze zneužít k odstranění ochrany proti clickjackingu a to včetně případu, kdy je zabezpečení tvořeno také hlavičkou X-Frame-Options. Jedná se především o zranitelnosti zneužitelné k provedení změn v obsahu souborů na serveru nebo jednorázovému podstrčení vlastních skriptů, ale potenciálně může k tomuto cíli vést téměř jakákoli zranitelnost, protože v závislosti na webové aplikaci může každé zneužití určité zranitelnosti vést k otevření dalších slabých míst v zabezpečení webové aplikace. Riziko zde představuje například nezabezpečený upload souborů, se kterým se lze setkat poměrně zřídka a mnohem účinnější je zde manuální testování. Proto s detekcí této zranitelnosti v aplikaci nijak nepočítám.

V případě detekce dalších zranitelností počítám s určitým rozšířením do budoucna a to jednak z časových důvodů, méně častým výskytům těchto zranitelností, ale také kvůli předpokládaným problémům z hlediska spolehlivosti automatizované detekce. V tomto případě se jedná například o testování zabezpečení proti útokům SSI nebo PHP injection.

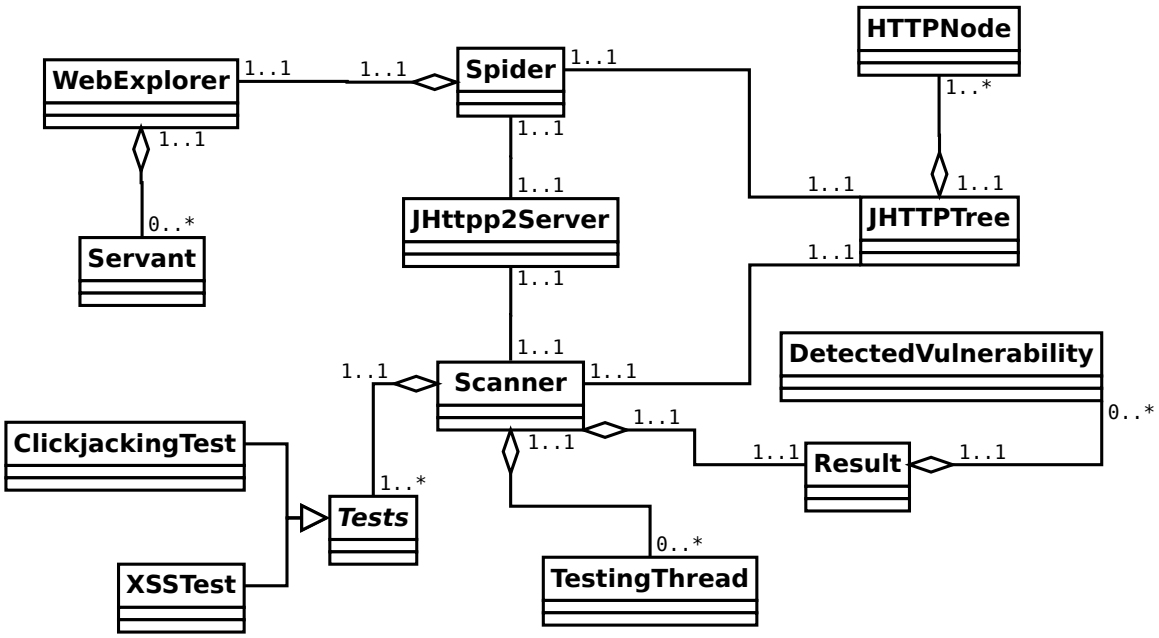
[1] Viz podsekce 3.4.1.
[2] Viz úvod sekce 4.3.
[3] Viz sekce 3.1.

Kapitola 6

Návrh

Na základě předchozí analýzy a z poznatků získaných nastudováním problematiky clickjackingu a penetračního testování jsem vytvořil návrh aplikace, kterou jsem pojmenoval WebSerpent. Z pohledu požadavků je patrné, že aplikace nemá být pouhou aplikací, která detekuje či nedetekuje určité zranitelnosti na stránce, ale že je koncipována jako nástroj, který má ulehčit proces celého testování.

Strukturu aplikace lze rozdělit do tří modulů: Proxy, Spider a Scanner, nad kterými je postaveno grafické uživatelské rozhraní. Základní achitektura aplikace je naznačena v třídním diagramu 6.1, který obsahuje kvůli přehlednosti pouze vybrané třídy bez návaznosti na GUI[1].



Obrázek 6.1: Diagram tříd aplikace WebSerpent znázorňující základní architektru aplikace.

[1] GUI – grafické uživatelské rozhraní. Viz sekce 6.4

6.1 Proxy

Proxy server je využíván ke zprostředkování veškeré HTTP(S) komunikace mezi prohlížečem a serverem nebo pro vytvoření SSL tunelu při testování nebo mapování webu. Při obdržení HTTP odpovědi od serveru je nejprve zpracována a posléze je zpracovaná kopie předána spolu s odpovídající HTTP žádostí modulu Spider. Zároveň je obdržená odpověď přeposlána dále prohlížeči. V rámci uživatelského rozhraní lze změnit port, na kterém naslouchá příchozí HTTP požadavky. Rozmezí povolených portů je od 1024 do 65535. Defaultně je využíván port 8088. Více viz sekce 7.1.

6.2 Spider

Tento modul je zodpovědný za sledování HTTP komunikace a mapování webu, které je prováděno třídami `WebExplorer` a `Servant`. Při obdržení HTTP žádosti a odpovědi jsou zde před přidáním do stromu `JHTTPTree` filtrovány. Uchovávána je pouze komunikace obsahující HTML, CSS nebo javascriptový kód, a která nepochází z URI, jejíž cesta nebo GET parametry obsahují řetězec zavedený v blacklistu[2].

Zavedený blacklist je využíván zejména při automatizovaném mapování webu, kdy není žádáno odhlášení uživatele z aplikace (pokud je přihlášen). Do stromu není stránka zavedena ani při manuálním navštívení (tedy odhlášení), aby nebyla nedopatřením zahrnuta do testování. Blacklist je rovněž uživatelem upravitelný.

6.2.1 Automatizované mapování

Při automatizovaném mapování jsou navštěvovány všechny stránky s výjimkou stránek zmíněných výše. Mapování je obstaráváno několika vlákny (třída `Servant`) dokud není prohledán celý web z množiny míst, které vybere uživatel před zahájením mapování. Mapování probíhá z každého místa až do houbky, kterou stejně jako počet vláken může uživatel nastavit.

Kromě vytváření mapy stránek zároveň vyhledává formuláře, které dosud nebyly odeslány. Protože by nemusely být automatickým vyplněním vyplněny validně, rozhodl jsem se pouze upozornit uživatele na jejich výskyt. Uživatel pak může stránku zobrazit v prohlížeči a korektně formulář vyplnit.

6.2.2 Speciální domény

V modulu je vedena rovněž speciální databáze s různými doménami, které si navolí uživatel. Defaultně jsou stránky v databázi ukládány na základě domény, cesty, metody a názvů parametrů. U speciálních domén jsou stránky ukládány také na základě hodnot jednotlivých parametrů. Toto je výhodné u webů, které povolují přístup pouze k jednomu souboru (např. index.php) a všechny ostatní stránky jsou skripty vkládány na základě získaných parametrů. Je tak možno otestovat najednou celý web. Což například v případě nástroje Paros, který tento přístup ukládání nepodporuje, v některých případech není možné.

[2] blacklist – doslova černá listina

6.3 Scanner

Scanner je modul zajišťující testování (skenování) vybrané množiny webů. V první fázi jsou vytvořeny a naplánovány veškeré testy pro jednotlivé stránky, které jsou zahrnuty ve zvolené testované množině stránek. Plánování samotných testů je ponecháno v režii testovacích tříd V druhé fázi jsou vytvořena testovací vlákna třídy `TestingThread`, kterými je prováděn samotný proces testování.

V případě nalezení zranitelnosti nebo při možnosti, že je stránka zranitelná (případ detekce clickjackingu) je výsledek testu uložen do třídy `DetectedVulnerability`, kde jsou uloženy všechny potřebné informace pro uživatele. Všechny výsledky jsou poté souhrnně uloženy v třídě `Result`. Ta se mimo jiné stará o vygenerování HTML reportu, který je přehlednější při průzkumu všech výsledků testů a lze jej použít například k přehledné prezentaci výsledků. Další prezentace výsledků je přímo skrze grafické uživatelské rozhraní (viz sekce 6.4).

Typy prováděných testů může uživatel opět nastavit. Ačkoli je apikace koncipována pro snadné začlenění dalších typů testů, v současnosti je k dispozici pouze testování clickjackingu a XSS. V následujících kapitolách je rozebrána detekce jednotlivých zranitelností.

6.3.1 Detekce zranitelnosti typu Clickjacking

V případě detekce zranitelnosti stránky proti útokům typu clickjacking, jsem se rozhodl vyhledávat bezpečnostní prvky na stránce, které by případnému pokusu o útok měly zabránit[3]. Mnou nalezené nástroje k detekci clickjackingu[4] byly závislé na prohlížeči a detekovaly až pokus o provedení samotného útoku. Metoda detekce bezpečnostních prvků, kterou jsem zvolil, je naopak zcela nezávislá na prohlížeči či grafické reprezentaci stránky. Další výhodou této metody je její rychlost, protože se zde nepracuje s grafickou prezentací stránky, ale jsou procházeny zdrojové kódy. Externí zdrojové kódy třetích stran, tj. z ostatních domén, nejsou procházeny.

Určitou nevýhodou je nemožnost úplného pokrytí všech různých variací *framebusteru*. V tomto případě ale předpokladám[5], že se bude většina vývojářů opakovat a většina stránek s tímto zabezpečením bude obsahovat již prezentovaná a známá řešení. Proto jsem se rozhodl detekovat tento kód v první řadě na základě testovací podmínky, pomocí regulárních výrazů, které zahrnují nejčastěji používané testovací podmínky (viz tabulka 6.1) a další jejich variace. V druhé řadě je obdobným způsobem vyhledána protiakce, která potvrdí či vyvrátí detekci framebusteru.

Zmíněné regulární výrazy jsou aplikovány až přímo na odpovídající části kódu, které jsou vyhrazeny konečným automatem. U nejčastěji vysktující se podmínky je tedy porovnáván pouze řetězec `top != self`. V případě shody je vybrán úsek kódu, který může být při splnění nebo nesplnění podmínky vykonán, a který je podroben druhé sadě regulárních výrazů určených k detekci protiakce.

Na základě detekováných prvků lze navíc do jisté míry spolehlivosti vyhodnotit předpokládanou úroveň zabezpečení, kterou framebuster poskytuje. Pokud pominu možnost výše zmíněné situace, kdy chybně není framebuster detekován, je obtížné deklarovat s jistotou,

[3] Framebuster kód a hlavička X-Frame-Options. Viz sekce 4.2.

[4] Viz podsekce 4.3.

[5] Tento předpoklad je založen na reakcích na různých internetových fórech, údajích z tabulky 6.1 a různých článcích, které se věnují clickjackingu a rovněž doporučují či zmiňují uvedené variace *framebasteru*.

Unikátních webů	Testovací podmínky
38%	`if (top != self)`
22.5%	`if (top.location != self.location)`
13.5%	`if (top.location != location)`
8%	`if (parent.frames.length > 0)`
5.5%	`if (window != top)`
5.5%	`if (window.top !== window.self)`
2%	`if (window.self != window.top)`
2%	`if (parent && parent != window)`
2%	`if (parent&&parent.frames&&parent.frames.length>0)`
2%	`if((self.parent&&!(self.parent===self))` `&&(self.parent.frames.length!=0))`

Tabulka 6.1: Přehled nejpoužívanějších podmínek u 500 nejnavštěvovanějších stránek dle žebříčku Alexa Top Sites z roku 2010. [11]

že daný framebuster je z hlediska stávajících znalostí neprolomitelný[6]. Z hlediska stávajícího návrhu jsem se rozhodl vyhodnotit nalezené framebustery až nižším ohodnocením odhadované úrovně zabezpečení (tzn. mimo nejvyšší stupeň), kdy stav lépe odpovídá situaci, že daný framebuster může být potenciálně naprosto bezpečný, ale vše je ponecháno na konečném vyhodnocení uživatelem. Uživatel zde má samozřejmě možnost změnit navržené ohodnocení u jednotlivých regulárních výrazů. V budoucnu je plánováno zavedení dodatečné analýzy CSS atributu `display` u HTML elementu `BODY` a detekce změny vlastnosti ve framebusteru, nezávisle na uživatelem ovlivnitelných regulárních výrazech. Problém zde představují různé možnosti, kterými může být této změny docíleno (například možnost smazání obsahu HTML elementu, který v sobě nastavení tohoto CSS atributu pro element `BODY` obsahuje), které vyžadují dynamickou detekci změny této vlastnosti.

Nedostatky, které jsou způsobeny nemožností pokrytí všech možných variací bezpečnostního kódu (přestože předpokládám, že lze výše popsaným způsobem detekovat vysoké procento z celkového počtu stránek obsahujících *framebuster*), částečně řeší možnost parametrizace odpovídajícího testu. V tomto případě bude mít uživatel možnost upravit množinu používaných regulárních výrazů dle vlastních potřeb. Při nalezení další používané variace, může uživatel zavést nový regulární výraz, který příště umožní detekci i této varianty bezpečnostního kódu. Každý takový regulární výraz má navíc přiřazeno určité ohodnocení, určující uroveň zabezpečení, do které daná variace spadá (např. slabé zabezpečení, silné zabezpečení). Případně zde je možnost přiřazení dodatečného hlášení pro případné další uživatele. Takto lze i méně znalého uživatele informovat o nedostatcích nalezené obrany a nabídnout vzorovou ukázku kvalitního zabezpečení.

Další problém této detekce by mohla představovat sémantika napsaného kódu, kdy nejsem bez sémantické analýzy schopen určit, zda se v daném případě opravdu provede větev podmínky či nikoli. Určitou základní orientaci lze samozřejmě získat, pokud bude striktně kontrolována shoda celé podmínky s regulárním výrazem. Pro širší detekci jsem ale zvolil vyhledávání na shodu regulárního výrazu s podřetězcem porovnávané podmínky. Takto jsem schopen detekovat i testovací podmínky, které jsou například rozšířeny o udělení výjimek pro určité domény, kde dané regulární výrazy a varianty kódu již nelze efektivně

[6]Nejsou zde počítány situace, kdy je tato obrana zněškodněna využitím jiných zranitelnosti. Viz následující podsekce 6.3.2.

kontrolovat.

Z tohoto důvodu jsem se rozhodl do reportu o výsledcích testů zahrnout také výpis úseku kódu, který je detekovám jako *framebuster* (pokud je detekován). Uživatel se tak bude moci rozhodnout o správnosti výsledného zabezpečení. Zde ale předpokládám, že vývojář webové aplikace bude dostatečně zodpovědný, aby otestoval funkčnost napsaného kódu (opomíjím zde nefunkčnost při různých útocích) a tato možnost bude sloužit zejména pro kontrolu při udělování výjimek spřateleným webům a kontrolu plánované protiakce.

Dalším vyhledávaným bezpečnostním prvkem je hlavička X-Frame-Options. Detekce této hlavičky bude probíhat automaticky při parsování HTTP odpovědi, kdy je do proměnné uložen parametr této hlavičky. Původně jsem také plánoval v případě podpory CSP[7] detekci direktivy `frame-ancestors`, ale protože se do standardu tato direktiva nedostala [1], odsunul jsem její detekci na dobu, kdy bude také jeho součástí.

6.3.2 Detekce zranitelnosti typu XSS

Detekce probíhá formou pokusu o injektáž vlastního neškodného scriptu na stránku. Obsah jednotlivých proměnných na testované stránce je postupně nahrazován různými XSS vektory[8]. Pro každou takovou záměnu je vytvořen a posléze odeslán nový HTTP požadavek. V obdržené HTTP odpovědi je vyhledán výskyt injektovaného kódu, kdy je v případě nálezu detekována zranitelnost typu XSS pro daný vstup na stránce.

Není zde detekováno perzistentní XSS, které zde přinášelo spoustu problémů z hlediska vyhledávání výstupu. Zaslaný XSS vektor může být načítán na jiné stránce, než která je získána v obdržené HTTP odpovědi. Prohledávání webů by z tohoto hlediska mohlo být zdlouhavé, zejména na různých internetových fórech. Rovněž přiřazení jednotlivě nalezených uložených XSS vektorů k odpovídajícím vstupům by vyžadovalo provádění úprav vektorů.

Uživatel může libovolně upravovat množinu XSS vektorů. Vyhledávání vektorů ve vrácené odpovědi je přitom prováděno dvěmi způsoby. V případě, že není zadáno pole pro vyhledávání vektoru v odpovědi, je v odpovědi vyhledáván přímo odesílaný vektor jako řetězec. V případě vyplnění pole pro vyhledávání u vektoru vyplněno, je na stránce provedeno vyhledávání na základě regulárního výrazu. To umožňuje například testování vektorů, které jsou určeny pro vkládání hodnoty atributu v HTML tagu, které by se za normálních okolností jevily pouze jako obyčejný text a jejich detekce mimo tělo HTML tagu by nenasvědčovala výskytu zranitelnosti XSS.

6.4 Grafické uživatelské rozhraní (GUI)

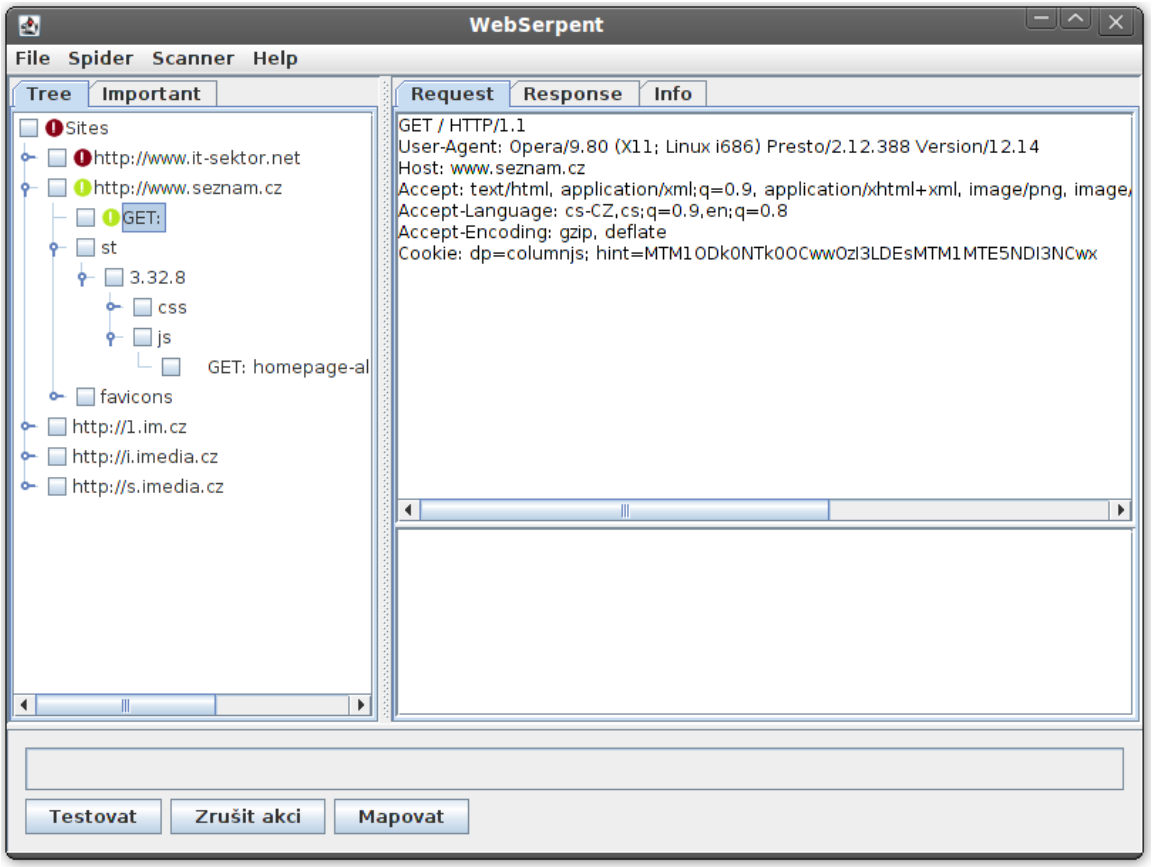
Při tvorbě GUI jsem se nechal do značné míry inspirovat nástrojem Paros, jehož GUI bylo přehledné a jednoduché. Na obrázku 6.2 lze vidět rozložení hlavního okna.

Část vlevo by se dala nazvat jako *přehled*. V kartě *Tree* lze vidět strom s *checkboxy*[9] s navštívenými weby, reprezentovaný třídou `JHTTPTree`. Ikony zobrazené u některých uzlů poskytují informaci o výsledcích posledního testu, který byl u nich proveden. Karta *Important* obsahuje seznam upozornění pro uživatele. Například jsou zde zasílány informace při potížích během testování nebo upozornění na nalezené nevyplněné formuláře během při automatizovaném mapování webu.

[7] Viz 3.3

[8] XSS vektor – řetězec vytvořený za účelem injektáže proveditelného skriptu na stránce.

[9] checkbox – zaškrtávací tlačítko

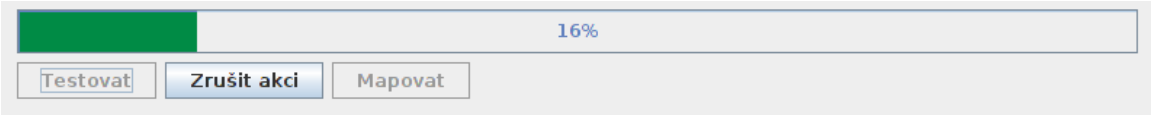


Obrázek 6.2: Hlavní okno aplikace WebSerpent. U vybraných uzlů stromu vlevo lze vidět informace o výsledích již proběhlých testů.

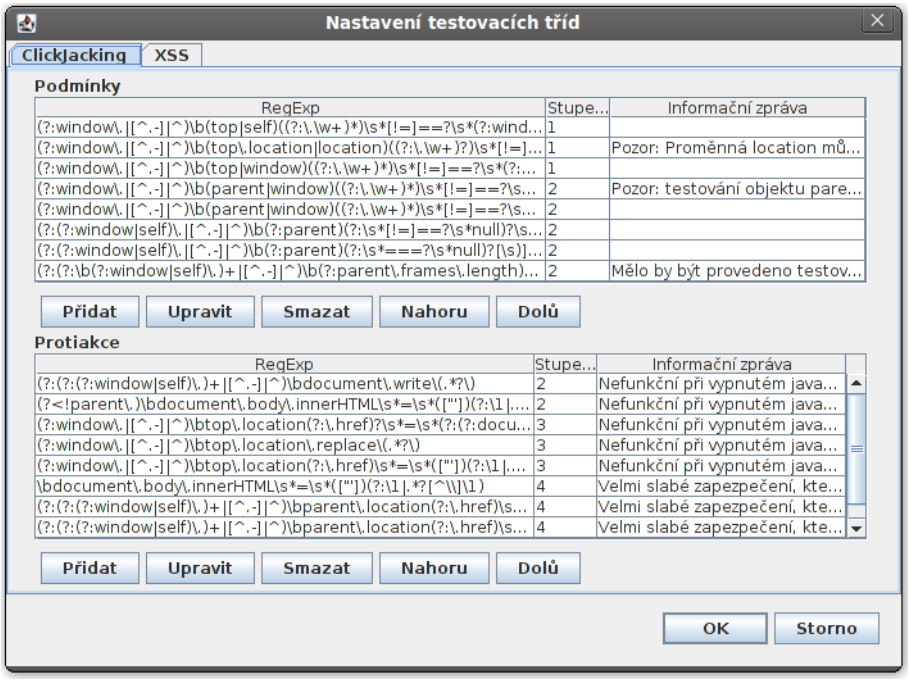
Část vpravo by se dala souhrně nazvat *detail*. Jsou zde karty pro zobrazení HTTP žádosti (Request), odpovědi (Response) a take informací ohledně posledního proběhlého testu (Info) u zvoleného uzlu nebo upozornění v sekci. V případě karet *Request* a *Response* je patrné rozdělení na HTTP hlavičku (nahoře) a tělo zprávy (dole). Poměr mezi oblastmi lze upravit. Stejně tak lze upravit horizontální rozdělení mezi levou a pravou částí.

Spodní část okna představuje ovládací panel. Nachází se zde *progressbar*, který slouží k informování o stavu právě probíhající operace, a tlačítka pro volbu akce a její zrušení. Na obrázku 6.3 lze vidět spodní část okna během testování.

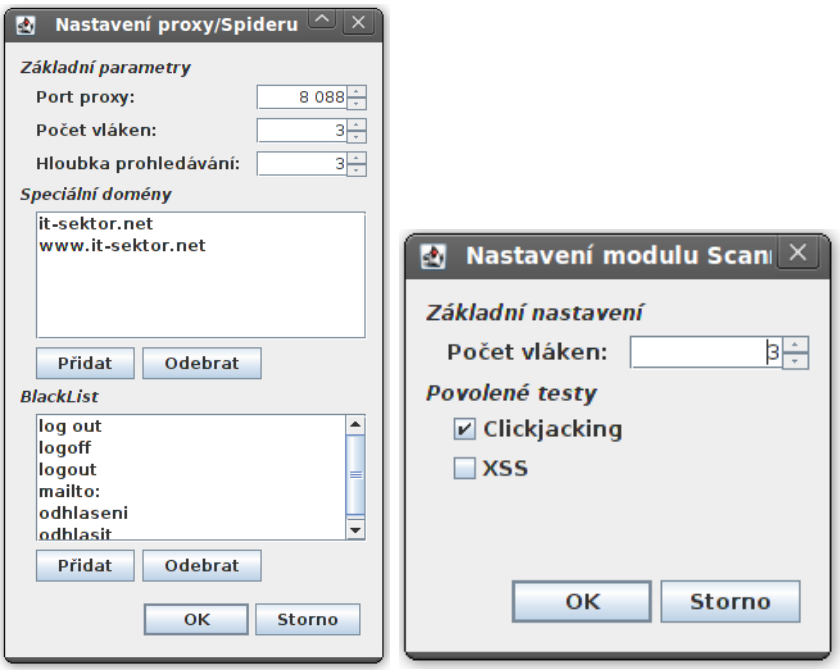
Skrze hlavní se pak lze dostat k různým nastavením aplikace, které lze vidět na obrázcích 6.4 a 6.5.



Obrázek 6.3: Funkce progressbaru během probíhajícího testování. Mezi aktivními tlačítky zůstává během operace pouze tlačítko ke zrušení akce.



Obrázek 6.4: Nastavení testovacích tříd. Zde lze parametrizovat testovaní webu na zranitelnost typu clickjacking.



Obrázek 6.5: Nastavení modulu Scanner (vpravo), Spider a Proxy (vlevo).

Kapitola 7

Implementace a testování

K implementaci aplikace jsem zvolil objektově orientovaný programovací jazyk Java verze 6, který byl v tomto případě favorizován již od začátku. Výhodou tohoto jazyka je široký výběr volně dostupných tříd a knihoven a platformová nezávislost. Dalším důvodem pro výběr tohoto jazyka byla již dřívější zkušenost s vytvářením grafického uživatelského prostředí.

7.1 Proxy server

K sestavení proxy serveru jsem využil dostupných opensource aplikací jHTTPp2[1] (HTTP proxy) a MITM[2] (HTTPS proxy). Servery, které dokázaly zprostředkovat jak HTTP tak HTTPS komunikaci, a které jsem nalezl, většinou nebyly opensource nebo byly využívány jinými penetračními nástroji, což mne od jejich využití odrazovalo. Zmíněné dvě aplikace byly opensource s dostatečně okomentovaným kódem a navíc byly miniaturní. Protože vyhovovaly mým plánům a při úvodním testování jsem neobjevil žádné problémy, rozhodl jsem se pro jejich nasazení.

HTTP i HTTPS požadavky jsou přijímány na stejném portu, který lze v nastavení změnit. Při HTTP komunikaci je využíváno pouze proxy serveru jHTTPp2. V případě obdržení HTTP požadavku s metodou CONNECT je komunikace předána proxy serveru MITM, který vytvoří SSL tunely s prohlížečem a serverem. Určitý problém zde nastal s používanými certifikáty, kdy je pro úspěsné vytvoření SSL tunelu v některých případech nutné přidat používanou certifikační autoritu do prohlížeče, popřípadě deaktivovat protokol pro online ověřování platnosti certifikátu (OCSP).

7.2 Parsování HTML

K parsování HTML kódu je využíván HTML parser Jericho[3]. Umožňuje spolehlivé a jednoduché nalezení různých HTML tagů v kódu, což značně zjednodušuje mapování webu nebo nalezení skriptů. Při parsování v současnosti představuje problém sytakticky chybně napsaná (X)HTML stránka. Jericho umožňuje provedení oprav určitých syntaktických chyb v kódu, nicméně jsem se rozhodl řešení tohoto problému odložil kvůli nízkému výskytu takto sepsaných webů (během nahodilého testování jsem narazil pouze na dva případy, kdy nebyla dodržena syntaxe (X)HTML kódu) a věnovat se závažnějším problémům.

[1] http://jhttp2.sourceforge.net/
[2] MITM – Man In The Middle. http://crypto.stanford.edu/ssl-mitm/
[3] http://jericho.htmlparser.net/docs/index.html

7.3 Uživatelské rozhraní

Kromě standardních tříd bylo při tvorbě uživatelského rozhraní využito implementace stromové struktury s checkboxy od Santhoshe Kumara[4], kterou jsem si upravil pro své potřeby. Určité problémy se vyskytly v závěru při dodatečných úpravách, při kterých byla k jednotlivým položkám přidána ikona informující o výsledku testu. Docházelo ke špatnému pozicování prvků v položce, kdy byla ikona schována za checkboxem nebo naopak byly umístěny mezi jednotlivými prvky příliš velké mezery. Pro nedostatek času jsem tuto chybu částečně vyřešil pevným umístěním ikony u listových položek, protože k problémům docházelo především při vkládání ikony mezi checkbox a název položky. I přesto ale občas k této chybě dochází, ale někdy stačí k odstranění této vady pouhé poklikání na položku. Přínos dodatečné grafické informace převažoval nad drobným estetickým nedostatkem a tak jsem tuto úpravu ponechal i ve stávající implementaci. V budoucnu plánuji kompletní přepis vykreslování prvků ve stromu, který by tento problém vyřešil.

7.4 Testování

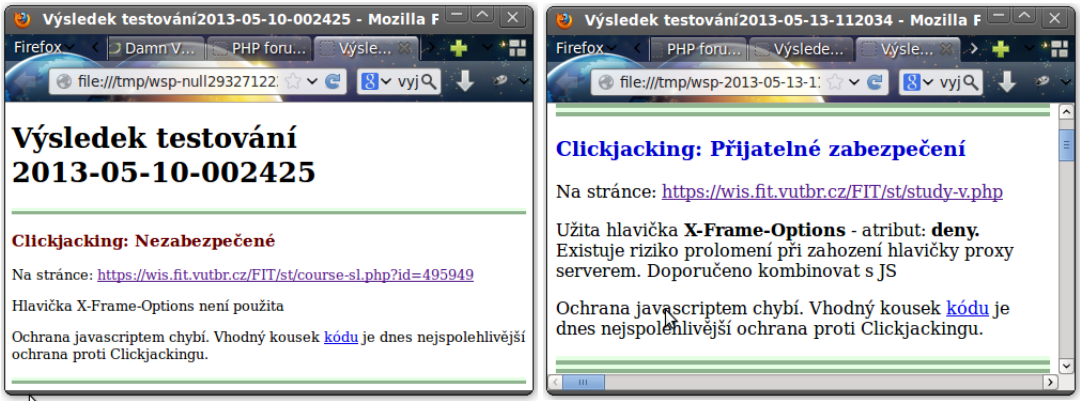
Aplikaci jsem průběžně testoval na náhodných internetových stránkách. Z hlediska detekce zranitelnosti XSS byl proveden test pouze lokálně na aplikaci DVWA[5], která slouží k těmto účelům, přestože se jedná pouze o základní test funkčnosti. Testování na ostrých aplikací jsem bez svolení majitelů nechtěl provádět a nenašel se nikdo, kdo by měl o otestování proti XSS útokům zájem. Test úspěšně nalezl na dané stránce zranitelnost typu XSS.

Obrázek 7.1: Report o nalezené XSS zranitelnosti.

V případě testování detekce zranitelnosti typu clickjacking je množina testovaných webů širší. Mezi testované weby patřil například také fakultní informační systém (WIS), jehož otestování prokázalo jeho zranitelnost. Web je v dnešních dnech již zabezpečen pomocí HTTP hlavičky X-Frame-Options. Přestože nepředpokládám, že by někdo ze studentů chtěl cíleně poškodit ostatní, například odhlášením projektů, potěšilo mne, že jsem tuto zranitelnost odhalil díky této práci.

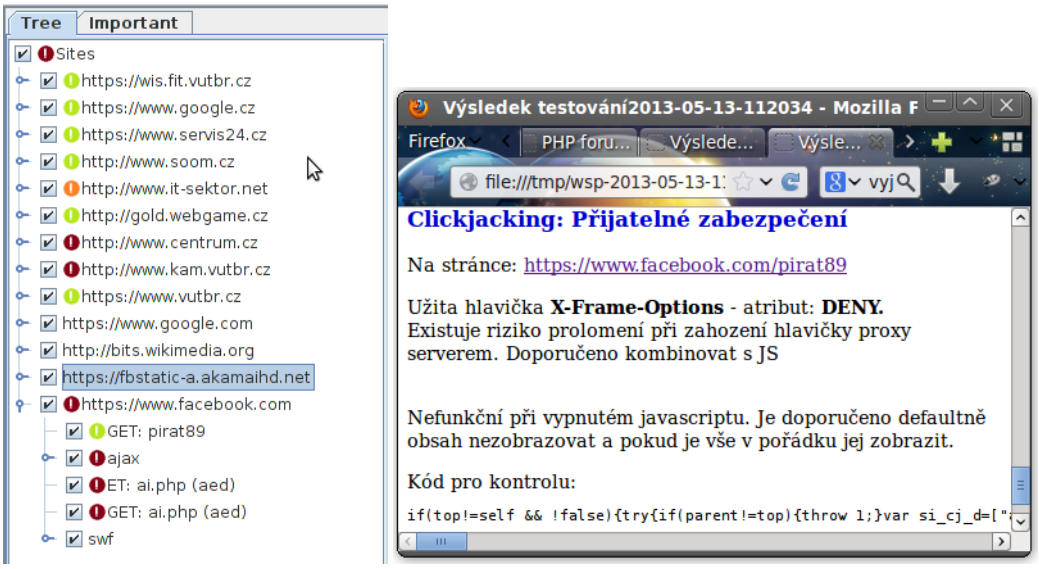
Testování na vybraných stránkách úspěšně nalezlo zabezpečení jak pomocí hlavičky X-Frame-Options, tak pomocí framebusteru. Výsledky testů ukázaly, že většina z náhodně

[4] http://www.jroller.com/santhosh/entry/jtree_with_checkboxes
[5] DWVA – Damn Vulnerable Web App



Obrázek 7.2: Zabezpečení WISu před opravou a po opravě.

vybraných webových aplikací spoléhá především na HTTP hlavičku X-Frame-Options. Zatímco framebuster je využíván jako záložní zabezpečení spíše výjimečně. Mezi aplikace používající framebuster v kombinaci s hlavičkou X-Frame-Options zde patří Facebook, který má zabezpečenu stránku osobního profilu uživatele.



Obrázek 7.3: Clickjecking detekce. Vlevo je strom s testovanými stránkami po provedení testu. Vpravo je výpis detekovaného framebusteru s hlavičkou X-Frame-Options

Kapitola 8

Závěr

Cílem této práce bylo vytvoření aplikace, která by detekovala zranitelnost typu clickjacking ve webových aplikacích. Tuto aplikaci jsem pojmenoval **WebSerpent**. Pro detekci zranitelnosti jsem zvolil metodu vyhledávání bezpečnostních prvků, které mají za úkol zabránit případnému clickjackingu.

Tým M. Balduzziho vytvořil aplikaci **ClickIDS** [2], která se zabývá rovněž automatizovanou detekcí clickjackingu. ClickIDS detekuje zranitelnost na základě úspěšného provedení útoku, ke kterému využívá vykreslovací jádro Gecko a je tak závislá na internetovém prohlížeči Mozzila Firefox. WebSerpent je naopak zcela nezávislý na používaném prohlížeči a díky detekci bezpečnostních prvků je schopen do jisté míry určit úroveň zabezpečení webu.

V rámci provedených testů (viz 7.4) byly úspěšně detekovány bezpečnostní prvky, na jejichž základě byl uživatel informován o předpokládané úrovni zabezpečení. Bohužel aplikace není nyní schopna spolehlivě identifikovat naprosto bezpečný framebuster, jehož identifikace spočívá nyní na rozhodnutí uživatele, kterému jsou ve výsledcích testů předloženy všechny podstatné informace o nalezených bezpečnostních prvcích.

Mezi testovanými weby byl také fakultní informační systém WIS, u nějž byla detekována zranitelnost typu clickjacking. Na základě této detekce byla zranitelnost ohlášena a následně byl systém zabezpečen pomocí HTTP hlavičky X-Frame-Options.

Oproti mému očekávání je zabezpečení pomocí framebusteru využíváno velmi zřídka. Většina aplikací, které byly zabezpečené proti clickjackingu, a které jsem nalezl, spoléhaly na zabezpečení pomocí výše zmíněné hlavičky. Předpokládám, že důvodem je jednoduchost vložení HTTP hlavičky do odpovědi a proti framebusteru mizivé riziko, že by se vývojář dopustil chyby. Na základě tohoto zjištění se domnívám, že podíl *exotických* framebusterů, které není aplikace bez úpravy testů v uživatelském prostředí schopna detekovat, budou pokrývat velmi malé procento z celkového počtu zabezpečených stránek.

Aplikace byla rozšířena také o detekci XSS zranitelnosti s výjimkou detekce perzistentního XSS. Ve srovnání s nástrojem Paros poskytuje tedy menší spektrum detekovatelných zranitelností, ale díky možnosti parametrizace jednotlivých testů může být testování v závislosti na znalostech uživatele důkladnější, například při obcházení XSS filtrů zabudovaných v aplikaci. Rovněž je zde možné díky speciálnímu módu pro zvolené domény otestovat kompletně celé weby, které využívají k navigaci pouze GET parametry. V případě aplikace Paros musí být testy spouštěny na jednotlivých stránkách.

Do aplikace byla zabudována dodatečná funkcionalita, která usnadňuje proces testování. Mj. automatizované mapování webů, možnost zvolit testovanou množinu webů napříč různými doménami, otevírat zachycené stránky v prohlížeči a možnost manuální detekce clickjackingu skrze testovací stránku.

8.1 Budoucí vývoj

Jak jsem naznačil již v předešlých kapitolách, mezi prvními náměty na další vývoj aplikace patří oblasti, které jsem z časových důvodů již nestihl dodělat. Mezi nimi je například odstranění občasných chyb ve vykreslování uzlů webového stromu (`JHTTPTree`). Dále přidání dodatečné detekce bezpečného framebusteru, kterou bude uživatel moci pouze aktivovat nebo deaktivovat, kvůli dodatečnému dynamickému vyhodnocení framebusteru nebo přidání možnosti povolení vyhledávání framebusteru u skriptů třetích stran. Přidání dalších typů testů nebo se věnovat detekci perzistentního XSS. V případě rozšíření normy CSP o direktivu `frame-ancestors` zavést detekci dalšího bezpečnostního prvku.

Za úvahu by stálo vyzkoušet kombinaci stávající detekce clickjackingu s detekcí „grafickou“, kdy by při nenalezení bezečnostních prvků byla provedena grafická detekce a v případě vyhodnocení stránky jako zabezpečené informovat uživatele o této zkutečnosti. Uživatel by poté mohl na základě této informace prohledat zdrojové kódy a aktualizovat databázi regulárních výrazů u stávající formy detekce. Mohlo by být využito nějakého emulatoru webového prohlížeče, aby aplikaace zůstala nadále nezávislá na prohlížeči. Nejprve by mohl být vytvořen obraz nevnořené stránky a ten následně porovnat s obrazem zanořené stránky. Problém by zde mohl být u dynamických a měnících se prvků. Jako například bannery, reklamy, atd. Nebo se vydat stejnou cestou, jako M. Balduzzi s jeho týmem [2].

Literatura

[1] *Content Security Policy* [online]. 2012 [cit. 20. února 2013]. Dostupné na: <http://www.w3.org/TR/CSP/>.

[2] Balduzzi, M., Egele, M., Kirda, E. et al. A solution for the automated detection of clickjacking attacks. In ACM. *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. New York, NY, USA: ACM, 2010. S. 135–144. ASIACCS '10. Dostupné na: <http://doi.acm.org/10.1145/1755688.1755706>. ISBN 978-1-60558-936-7.

[3] Berners Lee, T., Fielding, R. a Masinter, L. *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*. 2005. Dostupné na: <http://tools.ietf.org/html/rfc3986>.

[4] Fielding, R., Gettys, J., Mogul, J. et al. *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*. 1999. Dostupné na: <http://tools.ietf.org/html/rfc2616>.

[5] Hanson, R. R. *Clickjacking details* [online]. 2008 [cit. 20. února 2013]. Dostupné na: <http://ha.ckers.org/blog/20081007/clickjacking-details>.

[6] Huang, L.-S. a Jackson, C. *Clickjacking Attacks Unresolved* [online]. 2011 [cit. 20. únor 2013]. Dostupné na: <https://docs.google.com/document/pub?id=1hVcxPeCidZrM5acFH9ZoTYzg1D0VjkG3BDW_oUdn5qc>.

[7] Kristol, D. a Montulli, L. *HTTP State Management Mechanism* [RFC 2109 (Historic)]. únor 1997. Obsoleted by RFC 2965. Dostupné na: <http://www.ietf.org/rfc/rfc2109.txt>.

[8] Kümmel, R. *XSS: Cross-Site Scripting v praxi : o reálných zranitelnostech ve virtuálním světě*. [b.m.]: Tigris, 2011. Dostupné na: <http://books.google.cz/books?id=QMBetwAACAAJ>. ISBN 9788086062341.

[9] Law, E. *Combating ClickJacking With X-Frame-Options* [online]. 2010 [cit. 20. února 2013]. Dostupné na: <http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx>.

[10] Maone, G. *NoScript* [online]. 2012 [cit. 20. února 2013]. Dostupné na: <http://noscript.net/faq>.

[11] Rydstedt, G., Bursztein, E., Boneh, D. et al. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *In IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*. 2010. Dostupné na: <http://seclab.stanford.edu/websec/framebusting/framebust.pdf>.

[12] Tang, S., Dautenhahn, N. a King, S. T. Fortifying web-based applications automatically. In ACM. *Proceedings of the 18th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2011. S. 615–626. CCS '11. Dostupné na: <http://doi.acm.org/10.1145/2046707.2046777>. ISBN 978-1-4503-0948-6.

[13] Zalewski, M. *Browser Security Handbook* [online]. 2008 [cit. 20. února 2013]. Dostupné na: <http://code.google.com/p/browsersec/wiki/Part2>.